

Low-Latency Layer-Aware Proactive and Passive Container Migration in Meta Computing

Mengjie Liu^{2,1}, Yihua Li⁵, Fangyi Mou³, Zhiqing Tang¹, Jiong Lou⁴, Jianxiong Guo^{1,3}, and Weijia Jia^{1,3}

¹Institute of Artificial Intelligence and Future Networks, Beijing Normal University, China

²Faculty of Arts and Sciences, Beijing Normal University, China

³Guangdong Key Lab of AI and Multi-Modal Data Processing, BNU-HKBU United International College, China

⁴Department of Computer Science and Engineering, Shanghai Jiao Tong University, China

⁵Department of Informatics, University of Zurich, Zurich, Switzerland

mengjieliu@mail.bnu.edu.cn, yihua.li@uzh.ch, moufangyi@uic.edu.cn,

zhiqingtang@bnu.edu.cn, lj1994@sjtu.edu.cn, {jianxiongguo, jiawj}@bnu.edu.cn

Abstract—Meta computing is a new computing paradigm that aims to efficiently utilize all network computing resources to provide fault-tolerant, personalized services with strong security and privacy guarantees. It also seeks to virtualize the Internet as many meta computers. In meta computing, tasks can be assigned to containers at edge nodes for processing, based on container images with multiple layers. The dynamic and resource-constrained nature of meta computing environments requires an optimal container migration strategy for mobile users to minimize latency. However, the problem of container migration in meta computing has not been thoroughly explored. To address this gap, we present low-latency, layer-aware container migration strategies that consider both proactive and passive migration. Specifically: 1) We formulate the container migration problem in meta computing, taking into account layer dependencies to reduce migration costs and overall task duration by considering four delays. 2) We introduce a reinforcement learning algorithm based on policy gradients to minimize total latency by identifying layer dependencies for action selection, making decisions for both proactive and passive migration. Expert demonstrations are introduced to enhance exploitation. 3) Experiments using real data trajectories show that the algorithm outperforms baseline algorithms, achieving lower total latency.

Index Terms—Meta computing, reinforcement learning, task scheduling, container migration

I. INTRODUCTION

Mobile Edge Computing (MEC) offers substantial compute and storage resources at the network edge to meet Quality of Service (QoS) standards of mobile devices [1]. Nevertheless, the lack of trust among edge nodes hinders the consolidation of all computing resources, leading to the emergence of the computing power islands issue [2]. Meta computing is suggested as a solution to these issues, presenting a new computing paradigm with zero-trust-based computing modules [2]–[5]. By integrating cloud, edge, and other resources and virtualizing networks as meta computers, the management of

underlying resources is made transparent to end-users. Services can be deployed through containers on meta computers. When taking into account the user's mobility and the MEC server's limited coverage, the service can be dynamically migrated to a more appropriate MEC server and automatically scaled according to resource demand.

However, challenges arise when making container migration decisions in meta computing. Firstly, there is a need to fully utilize the characteristics of containers to minimize migration cost. Image files must be available locally before running the container [6], containing code, binaries, system tools, configuration files, etc. Otherwise, they need to be downloaded from the registry [7]. Current methods such as Slacker [8], Cntr [9], and Pocket [10] aim to reduce startup delays by making significant changes to the application, potentially diminishing the advantages of container isolation. These methods overlook the fact that each image is comprised of layers [11], which can be shared among multiple images. Researchers have introduced layer matching algorithms for container placement and migration [6], [12], [13]. However, the heterogeneity of edge nodes and user mobility in meta computing lead to complex and sparse layer sharing patterns among different tasks. These layer-specific features are both numerous and sparse. Developing a new method for extracting layer features poses a significant challenge that needs to be addressed.

The second challenge is how to make proactive and passive migration decisions jointly. Both proactive migration and passive migration are necessary in meta computing. Proactive migration involves migrating a container to other nodes based on the node resource status [14]. Goanon *et al.* [15] propose an architecture that ensures the monitoring system can proactively maintain QoS while managing resource allocation and scaling proactively. Tran *et al.* [16] propose a framework that can use application-specific knowledge to determine what data should be proactively migrated to minimize latency and optimize the use of potentially limited resources. Passive migration involves migrating mobile users beyond the communication range, and relocating services based on the user's movement. The concept of companion fog computing is introduced to migrate the fog

This work is supported in part by the Chinese National Research Fund (NSFC) under Grant 62302048 and 62272050; in part by Guangdong Key Lab of AI and Multi-modal Data Processing, UIC under Grant 2020KSYS007; in part by Zhuhai Science-Tech Innovation Bureau under Grant 2320004002772; and in part by Interdisciplinary Intelligence SuperComputer Center of Beijing Normal University (Zhuhai). (Corresponding author: Zhiqing Tang.)

service to always be close enough to the served device [17]. In meta computing, it is crucial to ensure that the service can only be performed within the topological distance from the edge node. Thus, making migration decisions remains a challenge.

To make migration decisions, several existing works [18]–[22] propose migration solutions using Markov Decision Process (MDP) [23] or Lyapunov optimization [24] assuming complete system information is known. Meanwhile, Deep Reinforcement Learning (DRL) [25] emerges as a promising approach for tackling continuous decision-making challenges. For instance, Lee *et al.* [26] introduce a DRL-based algorithm considering CPU, memory, disk, and packet loss utilization to determine migration actions. However, these studies insufficiently account for layer features and fail to address proactive and passive migration decisions.

In this paper, we present a layer-aware Proactive and Passive Container Migration (PPCM) algorithm utilizing proximal policy optimization (PPO) [27]. To capture layer sharing features effectively, we propose a feature extraction network based on Deep & Cross Network (DCN) [28]. DCN is capable of learning low-dimensional feature crossovers and high-dimensional nonlinear features efficiently compared to other machine learning methods, without the need for manual feature engineering. Additionally, we incorporate expert experience to expedite the training process of the algorithm and enhance the optimization of migration decisions. Experimental results demonstrate the superior performance of our algorithm over baseline algorithms.

The contributions of this paper are summarized as follows:

- 1) We introduce the container migration problem in meta computing for the first time. To efficiently reduce migration and computation costs, we thoroughly incorporate layer sharing information in our system model. We employ a DCN-based network for feature extraction to capture sparse layer sharing features.
- 2) For proactive and passive migration decisions, we introduce a PPO-based PPCM algorithm. Leveraging layer sharing features from DCN, this algorithm incorporates expert knowledge to enhance learning speed and decision-making efficiency, ultimately optimizing migration costs in the long run.
- 3) Experiments are conducted using real data sets. Based on the results, our algorithm shows significant performance improvements compared to traditional image-based algorithms and layer sharing-based heuristic algorithms.

The paper is structured as follows. Section II introduces the related work. Section III and Section IV presents the system model and problem formulation. Section V provides a description of the PPCM algorithm. Section VI conducts an experimental evaluation, and Section VII conclude the paper.

II. RELATED WORK

A. Service Migration

When users transition between adjacent or overlapping regions, service migration must consider both the migration

cost and the QoS expectations of the migrated users. With a large number of mobile users, diverse edge server types, and the intricate interplay between migration cost and transport, finding an optimal strategy poses challenges. Mirkin *et al.* [29] introduced a checkpointing and restarting feature for containers in OpenVZ, enabling the inspection and restarting of running containers on the same or different hosts. Docker has streamlined the rapid deployment of Linux applications in recent years [30]. Machen *et al.* [31] presented a layered framework for migrating running applications enclosed in virtual machines or containers, demonstrating the superior performance of containers over virtual machines. Wang *et al.* [32] proposed an online approximation algorithm with polynomial time complexity for real-time container-based service migration in a dynamic MEC environment. Montero *et al.* [33] advocated for container-based service migration using software defined networking and network function virtualisation to enhance security by relocating security applications from end-user devices to trusted network nodes at the network edge. Notably, all the aforementioned studies focus on service migration with containers but overlook the granular storage of images on layer units, neglecting layer granularity in service migration.

B. Layer-aware Scheduling

Scheduling at layer granularity shows potential. Tang *et al.* [13] suggested that utilizing layer-sharing data among containers for multiple users can notably cut migration latency. They introduced an innovative online container migration algorithm to lessen overall task latency. Gu *et al.* [34] explored layer-aware microservice placement and request scheduling at the edge, enhancing throughput and hosted microservice volume by sharing layers among nearby images. By sharing layers between these images, the throughput and number of hosted microservices can be significantly boosted. Smet *et al.* [35] introduced a heuristic for optimal layer placement, where clients download necessary layers from storage to the server to maximize demand within storage and latency limits. Darrous *et al.* [36] used K-centre optimization for placing container layers in edge nodes, arranging layers by size to minimize maximum retrieval time of container images. Ma *et al.* [37] leveraged the hierarchical storage system to reduce filesystem sync overhead and proposed a protocol for real-time migration of edge services for mobile users. Dolati *et al.* [38] presented a polynomial time algorithm based on deterministic and random rounding frames, addressing container layer download paths and locations, VNF chaining during network traffic routing, and orchestrating containerized VNF chains in the edge network. However, none of the above works consider the sharing of information between layers during proactive and passive container migration.

III. SYSTEM MODEL

A. System Model

Consider a diverse multi-user meta computing system. A group of mobile users are distributed in a specific area

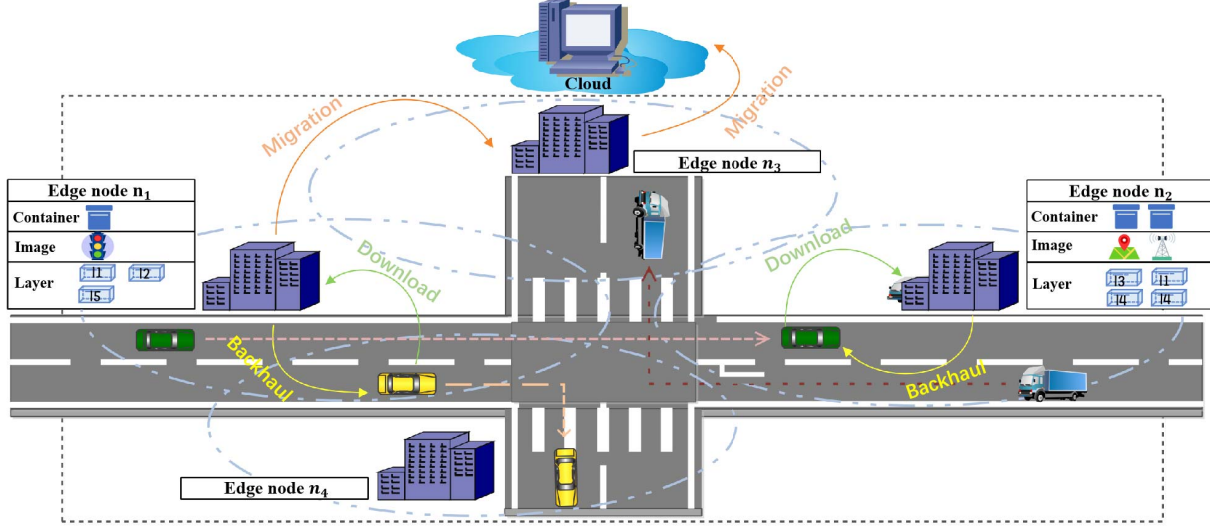


Fig. 1: Example of container migration in meta computing.

$\mathbf{U} = \{u_1, u_2, \dots, u_{|\mathbf{U}|}\}$, where $|\mathbf{U}|$ represents the set's size. In this meta computing setup, mobile users can offload their computational tasks to edge node services. Operating on a time-slice model, where a user's location changes only at the start of each time-slice, we can view the issue as a sampling in a continuous time model at various time points $\mathbf{T} = \{t_1, t_2, \dots, t_{|\mathbf{T}|}\}$, with a collection of user-generated computational tasks denoted as $\mathbf{K} = \{k_1, k_2, \dots, k_{|\mathbf{K}|}\}$. The edge node set is denoted as $\mathbf{N} = \{n_1, n_2, \dots, n_{|\mathbf{N}|}\}$, positioned at the edge of the core network. The cloud is treated as an edge node $n_{|\mathbf{N}|+1}$ with limitless computing resources. Each node possesses its CPU frequency f_n , bandwidth b_n , and storage space d_n . For a cloud node, the CPU frequency is f_{cloud} and the bandwidth is b_{cloud} . To handle the tasks, a variety of containers $\mathbf{C} = \{c_1, c_2, \dots, c_{|\mathbf{C}|}\}$ are established and deployed on the edge nodes. Image files are necessary to execute the containers, denoted as $\mathbf{M} = \{m_1, m_2, \dots, m_{|\mathbf{M}|}\}$. The images comprise the relevant layers, denoted as $\mathbf{L} = \{l_1, l_2, \dots, l_{|\mathbf{L}|}\}$, with each layer $l \in \mathbf{L}$ having a size d_l .

The set of layers in container $c \in \mathbf{C}$ is denoted as $\mathbf{L}_c = \{x_c^l | l \in \mathbf{L}\}$, where $x_c^l = 1$ indicates that container c holds layer l , otherwise $x_c^l = 0$. For each task $k \in \mathbf{K}$ created by the user at time t , the CPU resource requested is p_k and the designated container is c_k . Following scheduling, the node allocated to this task is represented as $\mathbf{n}_k = \{u_k^n | n \in \mathbf{N} \cup \{n_{|\mathbf{N}|+1}\}\}$, where $u_k^n = 1$ implies task k is assigned to node n , otherwise $u_k^n = 0$.

B. Cost

In a meta computing system, a mobile user u requests service from an edge node at time slice t , denoted as u_t^k . Generally, servers are interconnected through stable backhaul links, enabling mobile users to access services via multi-hop communication between edge nodes even when not directly

connected to the serving edge node. For optimal outcomes, services should dynamically migrate between nodes.

Migration is split into two parts: proactive migration and passive migration. Proactive migration is triggered when the current node lacks CPU and memory resources, leading to task reclamation and queuing. Resources occupied by the task on the node are then freed. Passive migration occurs when a mobile user surpasses a node's communication range, exceeding it compels the user to switch to another node. If no node meets the criteria during migration, the task is migrated to the cloud. Tasks move dynamically between nodes incurring delays, with the final task time comprising migration delay, computation delay, deployment delay, and backhaul delay.

Migration delay: Migration delay happens when a task moves from one node to another. It has three components. Firstly, the time of movement, i.e., the delay during migration, is defined as:

$$M(u, d_t) = \begin{cases} 0, & \text{if } d_t = 0, \\ \frac{s_u^s(t)}{\eta_t^{\text{migr}}} + \sigma_t^{\text{migr}} d_t, & \text{if } d_t \neq 0, \end{cases} \quad (1)$$

where d_t represents the distance between the current node u_t^k and the previous node u_{t-1}^k , σ_t^{migr} is a positive coefficient, η_t^{migr} denotes the network bandwidth on the migration path, $s_u^s(t)$ signifies the size of the migrated service. If $d_t = 0$, it implies no migration will take place, resulting in a migration delay of 0. Otherwise, if migration occurs, $M(u, d_t) = 0$.

Next, the download duration post migration is a crucial consideration. Each node features a layered download queue. When a new layer requires downloading, it joins this queue and might have to await the completion of another layer. Consequently, the download time for task k equates to the maximum completion time for all the necessary layers. The introduction of z_i^n serves to represent the completion time

for layer l on node n . If the layer is already present or hasn't commenced downloading, then $z_l^n = 0$. Otherwise, it is represented as:

$$z_l^n = \frac{d_l^u}{b_n}, \quad (2)$$

where d_l^u represents the layer size needed by user u , b_n stands for the node's bandwidth. In the case of a cloud node, $b_n = b_c$. Hence, the ultimate download time after migration can be formulated as:

$$D(u) = \max_{l \in \mathbf{L}} (z_l^n(t) \times x_c^l). \quad (3)$$

Third, await execution time. If a task requires migration, it is placed back in the task queue, waiting for selection of a new node, incurring additional wait time. Let the current time slice be t and the time slice entering the task queue be t_u , the calculation is:

$$w(u) = t - t_u. \quad (4)$$

Movement and post-migration downloads on each node can happen concurrently, so we only need to consider the maximum of both. The final migration delay can then be calculated as:

$$S = \max(M, D) + \sigma_t^{\text{wait}} w, \quad (5)$$

where σ_t^{wait} is a positive coefficient.

Computation delay: Mobile users assign computational tasks to nodes for processing, which incurs computation time. At time slice t , the task size $s_u^k(t)$ is offloaded by user u , with the task's processing density denoted as κ . Consequently, the CPU cycles needed to process the offloaded task can be computed as $c_t = s_u^k(t)\kappa$. Simultaneously, the workload w_t of the node is defined as the multiplication of the total task sizes running at the node and the processing density, i.e., $w_t = s_u^n(t)\kappa$. The computational power of a node is specified as $f(n_t)$. For the cloud, $f(n_t) = f(c_t)$. It is assumed that resources are allocated proportionally across nodes, implying that tasks receive computational resources based on their CPU cycle requirements. Therefore, the computational latency of executing the task at time slice t can be determined as:

$$C(n_t) = \frac{c_t}{\left(\frac{c_t}{w_t + c_t} f(n_t)\right)} = \frac{w_t + c_t}{f(n_t)}. \quad (6)$$

Deployment delay: The deployment delay comprises two components. One is the access delay between the mobile user and the node, as defined:

$$A(u) = \frac{s_u^k(t)}{\rho_t}, \quad (7)$$

where $s_u^k(t)$ represents the offloading task size ratio for user u at time slice t , and ρ_t denotes the wireless uplink transmission rate from the mobile user to the node.

The other aspect is the time taken to deploy to the node for downloading, i.e., the download time after migration $D(u)$. As the mobile user's access and the deployment download can

happen concurrently, the deployment delay is the maximum of the two:

$$P = \max(A, D). \quad (8)$$

Backhaul delay: The backhaul delay is the delay between an edge node and the user, defined as:

$$B(u) = \frac{s_u^k(t)}{\eta_t^{\text{bh}}} + \sigma_t^{\text{bh}} y_t, \quad (9)$$

where y_t represents the distance from the node to the user, σ_t^{bh} is a positive coefficient, and η_t^{bh} denotes the network bandwidth for the return path.

Finally, the overall task duration is defined as:

$$T_k = S + C + P + B. \quad (10)$$

IV. PROBLEM FORMULATION AND ANALYSIS

Constraints: During migration, decisions must meet various conditions. Firstly, a restriction on the maximum number of containers a node can simultaneously execute is defined as:

$$|C_n(t)| \leq C_n, \quad \forall t, \forall n. \quad (11)$$

Moreover, every node is subject to constraints on storage resources and computational capabilities. In the event that, at time slice t , the current node's resources fall below a specified threshold, proactive migration becomes necessary. This limitation is defined as:

$$\begin{aligned} \sum_{l \in \mathbf{L}} (1 - y_n^l(t)) \times d_l &\leq \sigma^{\text{mem}} \times d_n, \quad \forall t, \forall n, \\ \sum_{l \in \mathbf{L}} f(n_t^l) &\leq \sigma^{\text{cpu}} \times f(n_t), \quad \forall t, \forall n, \end{aligned} \quad (12)$$

where $y_n^l(t)$ denotes if layer l is active on node n at time t . σ^{mem} and σ^{cpu} are both positive threshold control coefficients.

Third, each task can only be deployed to one node or the cloud, as follows:

$$\sum_{n \in \mathbf{N} \cup \{n|\mathbf{N}|+1\}} u_k^n = 1, \quad \forall k. \quad (13)$$

Problem formulation: Our aim is to minimize the total task time of the dynamic meta-computing system while meeting the constraints. Thus, the problem is formulated as:

$$\begin{aligned} \text{Problem 1: } \min T &= \sum_{k \in \mathbf{K}} T_k, \\ \text{s.t.} \quad &\text{Eqs. (11) - (13)}. \end{aligned} \quad (14)$$

Problem 1 is a complex bin-packing problem that is NP-Hard and can only be tackled heuristically. Traditional algorithms can only iteratively decide based on a set strategy at each time step, lacking insight into the influential factors between adjacent time steps and the ability to adjust strategy as per the meta computing system's dynamics. Making decisions iteratively on a time slice is also notably time-consuming. Problem 1 meets the MDP criteria by carefully selecting time slices where the first-order transfer probabilities of task resource requirements are quasi-static rather than uniformly spread over an extended period, while also possessing memoryless task and environment updates. In contrast, reinforcement learning algorithms can progressively enhance solutions through continual learning and optimization.

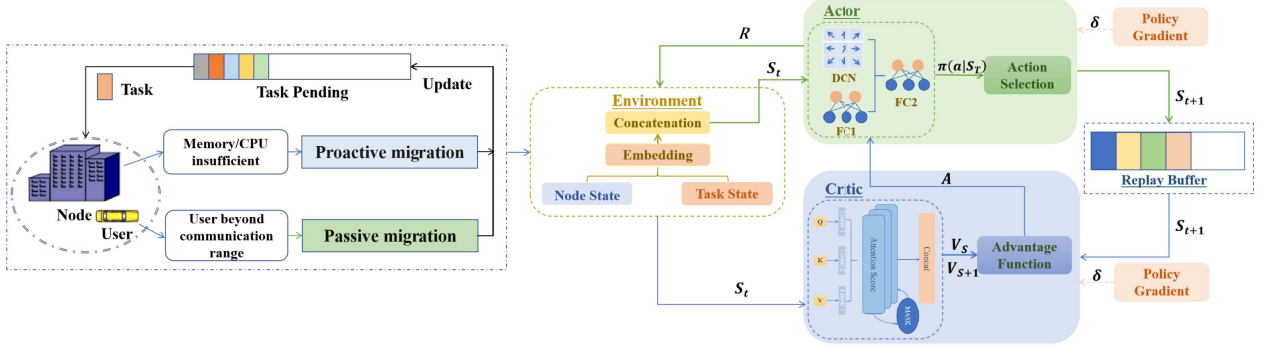


Fig. 2: Overview of the layer-aware proactive and passive container migration algorithm in meta computing.

V. OUR ALGORITHMS

A. Algorithm Settings

State: The state S_t provides a comprehensive overview of the meta computing, encompassing nodes and tasks.

The node state s_t^N contains the positional coordinates of the edge node $\{L_n^1(t) \dots L_n^{|N|}(t)\}$, comprising $\{x_n^1(t) \dots x_n^{|N|}(t)\}$ and $\{y_n^1(t) \dots y_n^{|N|}(t)\}$. It also includes the distance between the mobile user of the next task and the node $\{o_n^1(t) \dots o_n^{|N|}(t)\}$, along with the total size of the next task to be downloaded on the node $\{z_n^1(t) \dots z_n^{|N|}(t)\}$. The resource status of the node covers CPU resources f_n and bandwidth b_n , as well as the total size of all running tasks on the node $\{s_u^1(t) \dots s_u^{|N|}(t)\}$. Additionally, the state of each layer owned by the node $\{L_n^1(t) \dots L_n^{|N|}(t)\}$ is considered. Therefore, the final node state is represented as:

$$s_t^N = \left\{ \begin{array}{ccc} L_n^1(t) & \dots & L_n^{|N|}(t) \\ o_n^1(t) & \dots & o_n^{|N|}(t) \\ z_n^1(t) & \dots & z_n^{|N|}(t) \\ s_u^1(t) & \dots & s_u^{|N|}(t) \\ L_n^1(t) & \dots & L_n^{|N|}(t) \end{array} \right\} \cup \{f_n, b_n\}. \quad (15)$$

The task state s_t^k includes the location coordinates $L_u(t)$ of the mobile user, consisting of $x_u(t)$ and $y_u(t)$. It also encompasses the resources required for the task, such as CPU resources f_u and bandwidth b_u , along with the total download size $D_u(t)$ needed by the mobile user for the upcoming task. Additionally, it includes the layer state required for the next task $\{L_u^1(t) \dots L_u^{|L|}(t)\}$. Therefore, the final task state can be depicted as:

$$s_t^k = \left\{ L_u^1(t) \dots L_u^{|L|}(t) \right\} \cup \{L_u(t), f_u, b_u, D_u(t)\}. \quad (16)$$

The final representation of the state is:

$$s_t = s_t^N \cup s_t^k. \quad (17)$$

Action: The agent performs actions to assign task nodes, so the action space consists of all nodes and the cloud as follows:

$$a_t \in \mathbf{N} \cup \{n_{|\mathbf{N}|+1}\}. \quad (18)$$

Reward: The algorithm aims to maximize reward, while in meta computing, the objective is to minimize total task time, hence the reward is calculated as $r_t = -T_k$. From a long-term perspective, the cumulative reward is:

$$R(\tau) = \sum_{t=0}^T \gamma^t r_t, \quad (19)$$

where γ^t is a discount factor ranging from 0 to 1.

B. Expert Demonstrations

In traditional reinforcement learning, the agent seeks to uncover the optimal strategy for maximizing long-term rewards. Typically, the learning process commences with random exploration. However, as the exploration space expands exponentially in meta computing, particularly in scenarios with limited rewards, it can lead to ineffective learning. As suggested by [39], the PPCM algorithm proposes pre-training agents with expert demonstrations gathered through conventional means. This approach involves pre-training agents by first introducing a baseline algorithm based on expert demonstrations, storing the demonstration outcomes in an experience buffer to facilitate agent pre-training, initiating learning through imitation as a warm-start, and subsequently applying traditional reinforcement learning algorithms to optimize the pre-trained strategies. Simultaneously, to prevent excessive influence from the demonstrations on the learned policy, the number of demonstration samples is gradually reduced throughout the pre-training phase [26]. The details are outlined in the following section.

C. Migration Policy

The algorithm overview is depicted in Fig 2. The agent observes node and task states, embeds and connects them, then inputs into the policy network for action selection, migration, and reward generation. The policy network and value function are updated using a policy gradient-based algorithm.

Policy network: DCN [40] is employed in the policy network for extracting inter-dependencies within and across layers. This deep model efficiently learns low-dimensional feature crossovers and high-dimensional nonlinear features

Algorithm 1 PPCM

Require: Initialize policy parameters θ , replay memory $\mathcal{D}_i$, value function parameters ϕ , expert guidance threshold α , task pending $\mathcal{P}$

Ensure: Selected action a_t

```

1: for episode  $i \leftarrow 0, 1, 2, \dots$  do
2:   Initialize replay memory  $\mathcal{D}_i = \emptyset$ 
3:   Initialize task queue  $\mathcal{Q} = \emptyset$ 
4:   if  $i < \alpha_{\text{expert}}$  then
5:     for time step  $t \leftarrow 0, 1, 2, \dots$  do
6:       Select action  $a_t$  using expert policy
7:       Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{D}_i$ 
8:     end for
9:   end if
10:  for time step  $t \leftarrow 0, 1, 2, \dots$  do
11:    Observe current state  $s_t$ 
12:    if new task arrives and  $\mathcal{P}$  is not empty then
13:      for each task  $k$  in  $\mathcal{P}$  do
14:        if resource insufficient for  $k$  then
15:          Perform proactive migration for  $k$ 
16:        else if user beyond communication range for  $k$  then
17:          Perform passive migration for  $k$ 
18:        end if
19:        Update task status in  $\mathcal{P}$ 
20:      end for
21:    end if
22:    Select and execute action  $a_t$  using policy  $\pi_\theta(a_t|s_t)$ 
23:    Observe reward  $r_t$  and next state  $s_{t+1}$ 
24:    Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{D}_i$ 
25:  end for
26:  for training step  $j \leftarrow 0, 1, 2, \dots$  do
27:    Estimate advantages using Eq. (25)
28:    Compute policy update using Eq. (24)
29:    Update policy by maximizing Eq. (23)
30:  end for
31: end for

```

while demanding minimal computational resources. Network input features are categorized into dense and sparse features. The sparse feature input comprises the layer state of each edge node, while the dense feature input includes the other states. The cross network layer learns a low-dimensional cross combination of features, depicted as:

$$\mathbf{x}_{l+1} = \mathbf{x}_0 \mathbf{x}_l^T \mathbf{w}_l + \mathbf{b}_l + \mathbf{x}_l. \quad (20)$$

where $\mathbf{x}_l$ and $\mathbf{x}_{l+1}$ represent column vectors denoting the outputs from the l -th and $(l+1)$ -th cross layers; $\mathbf{w}_l$ and $\mathbf{b}_l$ denote the weight and bias parameters of the l -th layer. In the deep network section, a fully connected neural network is utilized to learn high-dimensional nonlinear feature cross combinations. The formula is as follows:

$$\mathbf{h}_{l+1} = f(W_l \mathbf{h}_l + \mathbf{b}_l). \quad (21)$$

where $\mathbf{h}_l$ and $\mathbf{h}_{l+1}$ represent the l -th and $(l+1)$ -th hidden layer; W_l and $\mathbf{b}_l$ are parameters for the l -th layer. $f(\cdot)$ denotes the ReLU function. Finally, the outputs of the cross network and deep network are merged to create the combination layer.

Value network: In value networks, we employ a multi-head attention mechanism [41] to assess state values. This mechanism can merge encoded representation information from various subspaces, analyze the initial input sequences using multiple self-attention sets, and subsequently integrate the outcomes of each self-attention set for a linear transformation to derive the final output results. It is obtained as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \\ \text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \quad (22)$$

where Q, K, V are query matrices, key matrices, and value matrices, respectively. $\text{head} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

Training: The PPCM algorithm is designed based on the PPO algorithm. PPO is built upon the policy gradient algorithm to guarantee efficiency with low computational complexity. Given π_θ as a policy with parameter θ , the optimization objective of PPO is as follows:

$$\theta_{k+1} = \arg \max_{\theta} \mathcal{L}^{PPO}(\theta_k, \theta). \quad (23)$$

The loss function for the policy gradient is defined as:

$$\mathcal{L}^{PPO}(\theta_k, \theta) = \mathbb{E}_{s, a \sim \pi_{\theta_k}} \left[\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \right. \right. \\ \left. \left. \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon\right) A^{\pi_{\theta_k}}(s, a) \right) \right], \quad (24)$$

where the function $\text{clip}(x, y, z) = \max(\min(x, z), y)$ restricts x to the interval $[y, z]$, where ϵ denotes the clip's range. Moreover, PPO employs a generalized dominance estimator for dominance calculation, which can be computed as follows:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \\ \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t), \quad (25)$$

where λ represents a General Advantage Estimation (GAE) parameter [42], δ_t signifies the temporal difference error at time step t , and V stands for an approximation function.

The PPCM algorithm is explained in Algorithm 1. Initially, the algorithm initializes the replay memory $\mathcal{D}_i$ for each time point (line 2). Expert demonstrations occur during pre-processing, and at each time step t , the expert strategy chooses the action and records the state transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}_i$ (lines 4-8). Then, at each time step t , the algorithm observes the current state s_t (line 11) and makes a migration type judgment if a new task arrives and the task queue is not empty (line 12). Proactive migration is carried out if the node has low resources (lines 14-15). Passive migration is executed if the user surpasses the communication range (lines 16-17). Following migration, a state update is performed (line 19). The action a_t is chosen based on the current policy

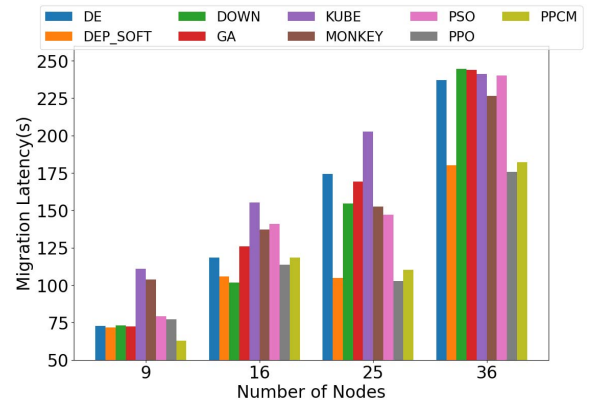
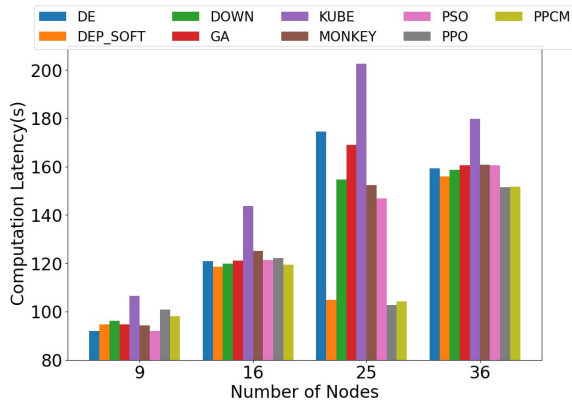
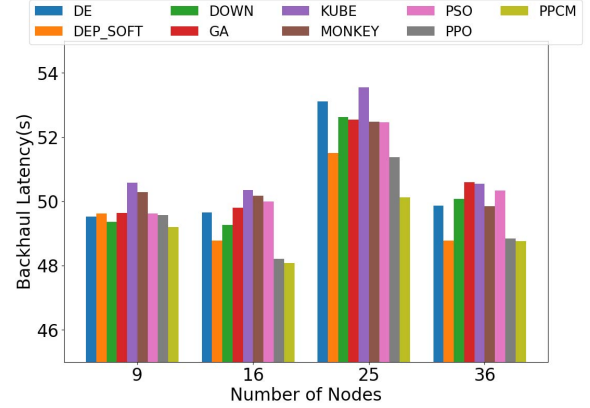
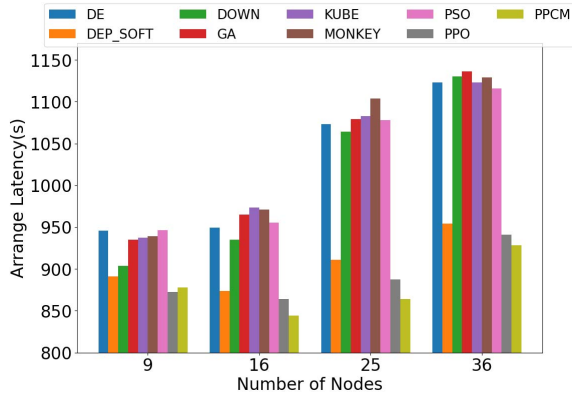


Fig. 3: Performance with different number of nodes

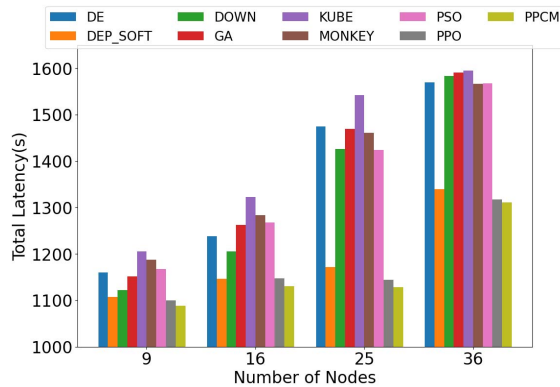


Fig. 4: Total latency

π_θ , executed, and the reward along with the next state are observed, and the state transition is stored in $\mathcal{D}_i$ (lines 22-24). Finally, during the training phase, for each training step j in each episode i , the algorithm utilizes the Eq. (25) to compute a policy update (lines 26-29). This update is accomplished by

maximizing the objective function using the stochastic gradient ascent algorithm of the Adam algorithm.

TABLE I: Hyperparameter Settings of PPCM Algorithm

Hyperparameter	Value	Hyperparameter	Value
Actor layer type	Dense	Critic layer type	Dense
Learning rate	0.0005	Training epoch	10
Batch size	512	Optimizer	Adam
Discount λ	0.95	Discount γ	0.99
Coefficient c_h	0.01	Clipping Value ϵ	0.2

VI. EVALUATION

A. Experimental Settings

We assess the algorithm using actual mobile trajectories in Rome, Italy [43]. We deploy 9 edge nodes in each region, each covering a $1 \text{ km} \times 1 \text{ km}$ grid with a computational power of 128 GHz. Typical real-world commercial 5G networks have upload rates of under 60 Mbps. Hence, in our scenario, the upload rate in each grid is set progressively as 60 Mbps, 48 Mbps, 36 Mbps, 24 Mbps, and 12 Mbps. The distance between two edge nodes is calculated using the Manhattan

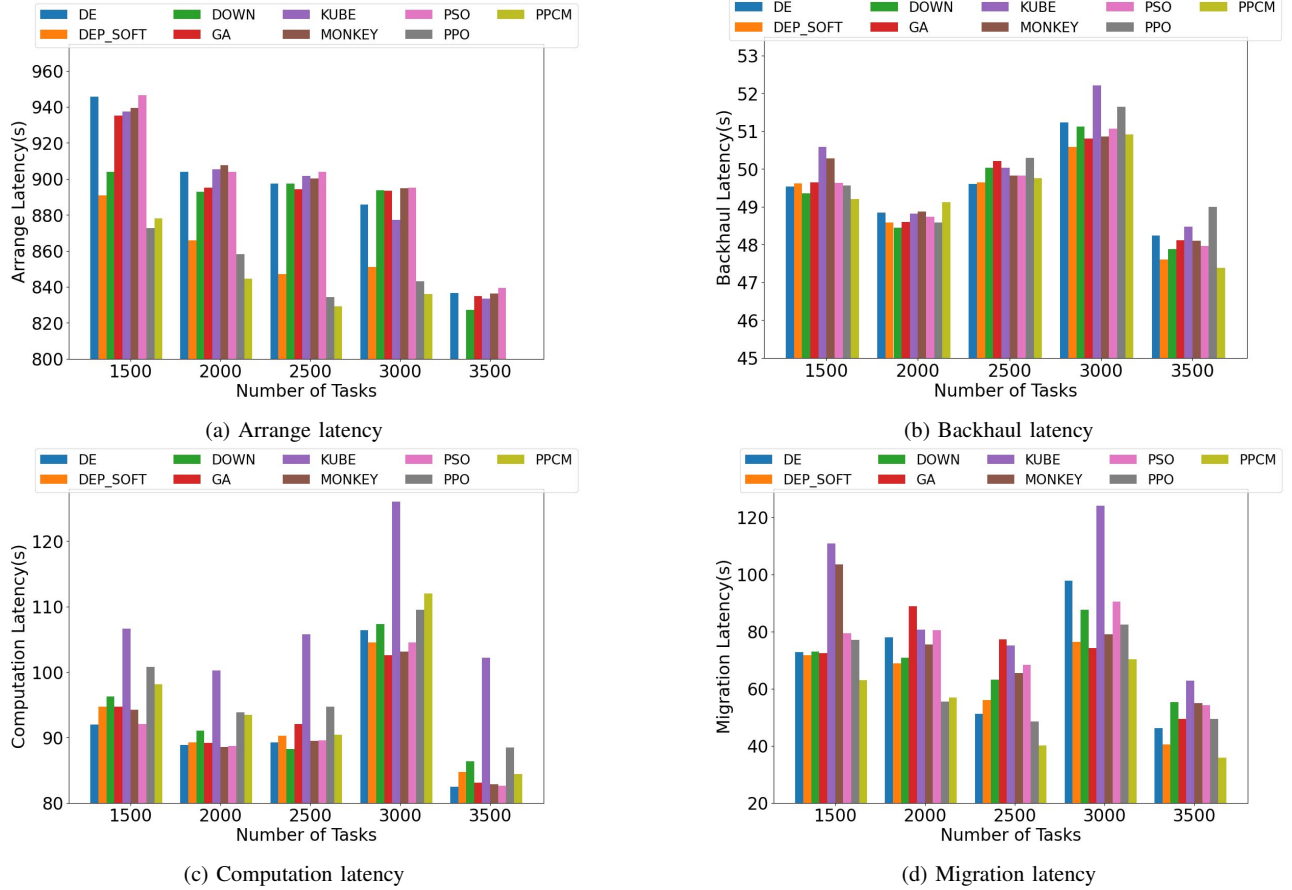


Fig. 5: Performance with different number of tasks

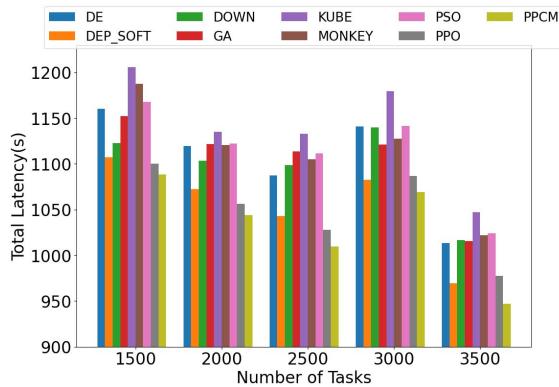


Fig. 6: Total latency

distance. Migration delay varies based on service sizes and network conditions. We assume the service size follows a uniform distribution in the range of $[0.5, 100]$ MB, and the migration delay factor follows a uniform distribution in the range of $[1.0, 3.0]$ s/hop during the training phase.

At each time step, tasks reaching the mobile user and the

edge node are randomly drawn from the Poisson distribution. According to empirical data, we evenly allocate the data size for each offloading task within the range of $[0.05, 5]$ MB, randomly select the service size from $[0.5, 100]$ MB, and uniformly distribute the processing density of the nodes within $[200, 10,000]$ cycles. Proactive migration is contingent on resource thresholds, where a positive factor of 0.2 of the resources acts as the triggering threshold, implying that migration occurs when less than 20% of the resources are available. Node resources are constrained to $[30, 100]$ GB, wired transmission bandwidth ranges from $[800, 1024]$ Mbps, backhaul network bandwidth is within $[400, 600]$ Mbps, and backhaul delay factor is fixed at 0.02 s/hop. The node can support a maximum of 30 containers. Besides, hyperparameters for the PPCM algorithm are outlined in TABLE I.

Baselines: Various baselines are conducted to assess performance. The action space for these algorithms includes both edge nodes and remote clouds. If any nodes lack sufficient resources, the task is offloaded to the cloud. The specifics are outlined below.

- 1) DEP_SOFT: It is a layer based scheduling algorithm [6]. Scores are calculated using local layer size. Simultane-

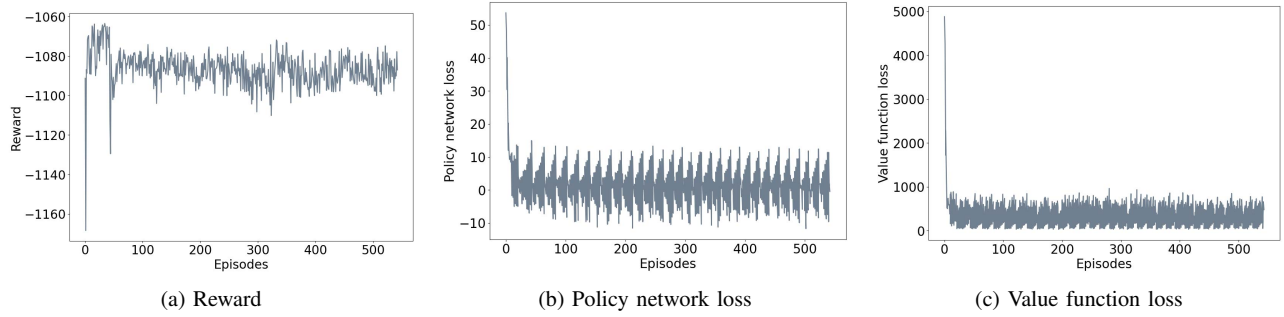


Fig. 7: Policy network Loss, value function Loss, and reward of the PPO-Expert algorithm

ously, a threshold is established, and a node is randomly chosen from nodes surpassing that threshold.

- 2) KUBE: This algorithm is image-based and serves as one of the default scheduling algorithms for Kubernetes. It evaluates the distribution of requested images across nodes and assigns a score according to available image sizes, determining scheduling based on this score.
- 3) MONKEY: A random algorithm. It picks a single node randomly for scheduling.
- 4) DOWN: When calculating the score, the size of the layer is changed to the estimated download time.
- 5) GA: It is a meta-heuristic algorithm inspired by the process of natural selection that relies on biologically inspired operations [6].
- 6) DE: Differential Evolution optimizes a problem by iteratively trying to improve candidate solutions with respect to a given quality metric [6].
- 7) PSO: Particle Swarm Optimisation is a meta-heuristic algorithm. It solves a problem where there is a swarm of candidate solutions and moves them in the search space according to a simple mathematical formula [6].
- 8) PPO: PPCM algorithm without expert demonstrations.

B. Experimental Results

Performance with different number of nodes. Figs. 3 and 4 illustrate the total delay as the number of nodes scales. The data indicates a noticeable trend: as the node number rises, the average task delay also increases. This phenomenon stems from the escalating cumulative delay caused by migrations. Consequently, the overall task delay grows with the expanding node number. In terms of total delay relationships, the hierarchy stands as follows: PPCM < PPO < DEP_SOFT < DOWN < PSO < DE < GA < MONKEY < KUBE, with potential minor deviations in the intermediary algorithms. Notably, PPCM consistently outperforms other algorithms irrespective of node number. Specifically, it achieves a reduction in total delay compared to all baseline algorithms by an average of 18%.

Performance with different number of tasks. As the number of tasks increases, task delay variation is depicted in Figs. 5 and 6. Results indicate that the PPCM algorithm exhibits superior performance. Among the baseline algorithms, DEP_SOFT consistently delivers the best results, while the

KUBE algorithm shows the worst performance. Overall, with an increasing number of tasks, the performance hierarchy based on total delay is PPCM < PPO < DEP_SOFT < DOWN < GA < DE < MONKEY < PSO < KUBE. Despite minor differences among the middle baseline algorithms, the PPCM algorithm reduces the total delay by an average of 7% compared to the baseline algorithms.

Performance of PPCM algorithm. Fig. 7 illustrates the convergence of the PPCM algorithm. The policy network loss and value function loss initially have high values during training. Nevertheless, with the progression of training steps, both decrease swiftly and stabilize around a specific value, signifying the algorithm's convergence.

VII. CONCLUSION

In this paper, we have modeled the layer-aware proactive and passive container migration problem in meta-computing. To efficiently reduce migration and computation costs, we thoroughly incorporate layer sharing information. For proactive and passive migration decisions, we introduce a PPO-based PPCM algorithm. We employ a DCN-based network for feature extraction to capture sparse layer sharing features. Expert knowledge is incorporated to enhance learning speed and decision-making efficiency, ultimately optimizing migration costs in the long run. Experimental results have shown the effectiveness of our proposed PPCM algorithm. Future work will consider the auto-scaling problem of containers in meta computing.

REFERENCES

- [1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
- [2] X. Cheng, M. Xu, R. Pan, D. Yu, C. Wang, X. Xiao, and W. Lyu, "Meta computing," *IEEE Network*, pp. 1–7, 2023.
- [3] M. Xu, S. Liu, D. Yu, X. Cheng, S. Guo, and J. Yu, "Cloudchain: a cloud blockchain using shared memory consensus and rdma," *IEEE Transactions on Computers*, vol. 71, no. 12, pp. 3242–3253, 2022.
- [4] H. Qi, M. Xu, X. Cheng, and W. Lyu, "Latency-first smart contract: Overclock the blockchain for a while," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.
- [5] Y. Li, J. Guo, Y. Li, T. Wang, and W. Jia, "An online resource scheduling for maximizing quality-of-experience in meta computing," *arXiv preprint arXiv:2304.13463*, 2023.

- [6] Z. Tang, J. Lou, and W. Jia, "Layer dependency-aware learning scheduling algorithms for containers in mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 22, no. 06, pp. 3444–3459, 2023.
- [7] A. Anwar, M. Mohamed, V. Tarasov, M. Little, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig *et al.*, "Improving docker registry design based on production workload analysis," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 265–278.
- [8] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 181–195.
- [9] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasicki, "Cntr: lightweight {OS} containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 199–212.
- [10] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.
- [11] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.
- [12] J. Lou, H. Luo, Z. Tang, W. Jia, and W. Zhao, "Efficient container assignment and layer sequencing in edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 02, pp. 1118–1131, 2023.
- [13] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Multi-user layer-aware online container migration in edge-assisted vehicular networks," *IEEE/ACM Transactions on Networking*, 2023.
- [14] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes," in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2022, pp. 263–265.
- [15] G. Marques, P. Senna, S. Sargento, L. Carvalho, L. Pereira, and R. Matos, "Proactive resource management for cloud of services environments," *Future Generation Computer Systems*, vol. 150, pp. 90–102, 2024.
- [16] P. Bellavista, A. Corradi, L. Foschini, and D. Scotece, "Differentiated service/data migration for edge services leveraging container characteristics," *IEEE Access*, vol. 7, pp. 139 746–139 758, 2019.
- [17] C. Puliafito, E. Mingozzi, C. Vallati, F. Longo, and G. Merlino, "Companion fog computing: Supporting things mobility through container migration at the edge," in *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2018, pp. 97–105.
- [18] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, 2019.
- [19] X. Zhou, S. Ge, T. Qiu, K. Li, and M. Atiquzzaman, "Energy-efficient service migration for multi-user heterogeneous dense cellular networks," *IEEE Transactions on Mobile Computing*, vol. 22, no. 2, pp. 890–905, 2021.
- [20] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.
- [21] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Joint resource overbooking and container scheduling in edge computing," *IEEE Transactions on Mobile Computing*, 2024.
- [22] H. Cui, Z. Tang, J. Lou, W. Jia, and W. Zhao, "Latency-aware container scheduling in edge cluster upgrades: A deep reinforcement learning approach," *IEEE Transactions on Services Computing*, 2024.
- [23] G. Shani, D. Heckerman, R. I. Brafman, and C. Boutilier, "An mdp-based recommender system," *Journal of machine Learning research*, vol. 6, no. 9, 2005.
- [24] A. Sootla and J. Anderson, "On existence of solutions to structured lyapunov inequalities," in *2016 American Control Conference (ACC)*. IEEE, 2016, pp. 7013–7018.
- [25] S. M. Dietz and A. C. Repp, "Decreasing classroom misbehavior through the use of drl schedules of reinforcement 1," *Journal of Applied Behavior Analysis*, vol. 6, no. 3, pp. 457–463, 1973.
- [26] D.-Y. Lee, S.-Y. Jeong, K.-C. Ko, J.-H. Yoo, and J. W.-K. Hong, "Deep q-network-based auto scaling for service in a multi-access edge computing environment," *International Journal of Network Management*, vol. 31, no. 6, p. e2176, 2021.
- [27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.
- [28] R. Wang, B. Fu, G. Fu, and M. Wang, "Deep & cross network for ad click predictions," in *Proceedings of the ADKDD'17*, 2017, pp. 1–7.
- [29] A. Mirkin, A. Kuznetsov, and K. Kolyshekin, "Containers checkpointing and live migration," in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 85–90.
- [30] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
- [31] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Migrating running applications across mobile edge clouds: poster," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016, pp. 435–436.
- [32] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1002–1016, 2017.
- [33] D. Montero and R. Serral-Gracià, "Offloading personal security applications to the network edge: A mobile user case scenario," in *2016 International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2016, pp. 96–101.
- [34] L. Gu, D. Zeng, J. Hu, B. Li, and H. Jin, "Layer aware microservice placement and request scheduling at the edge," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–9.
- [35] P. Smet, B. Dhoedt, and P. Simoens, "Docker layer placement for on-demand provisioning of services on edge clouds," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 1161–1174, 2018.
- [36] J. Darrou, T. Lambert, and S. Ibrahim, "On the importance of container image placement for service provisioning in the edge," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, 2019, pp. 1–9.
- [37] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Transactions on Mobile Computing*, vol. 18, no. 9, pp. 2020–2033, 2019.
- [38] M. Dolati, S. H. Rastegar, A. Khonsari, and M. Ghaderi, "Layer-aware containerized service orchestration in edge networks," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1830–1846, 2023.
- [39] Y. Qiu, Y. Zhan, Y. Jin, J. Wang, and X. Zhang, "Sample-efficient multi-agent reinforcement learning with demonstrations for flocking control," in *2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall)*. IEEE, 2022, pp. 1–7.
- [40] R. Wang, B. Fu, G. Fu, and M. Wang, "Deep & cross network for ad click predictions," in *Proceedings of the ADKDD'17*, ser. ADKDD'17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: <https://doi.org/10.1145/3124749.3124754>
- [41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [42] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.
- [43] L. Bracciale, M. Bonola, P. Loret, G. Bianchi, R. Amici, and A. Rabuffi, "Crawdad roma/taxi," 2022. [Online]. Available: <https://dx.doi.org/10.15783/C7QC7M>